University of Calgary
Department of Computer Science
CPSC 559, Winter 2023

Distributed System

Group #8
Ricky Bhatti
Justin Parker
Afshin Rahman
Aleksander Rudolf
Anshdeep Singh

April 10, 2023

Contents

1 Project Design

Project Design

Group #8

1.1 Background

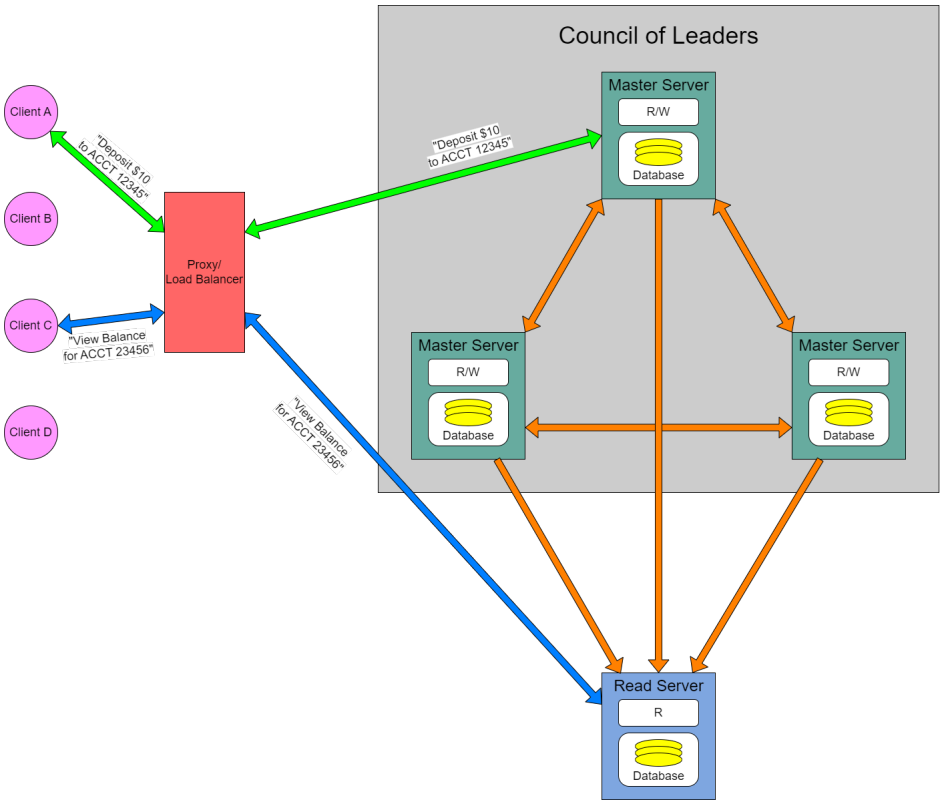
In the banking industry, billions of transactions occur daily at a high frequency. With transactions requiring absolute precision and little room for error, as small errors may result in the loss of billions of dollars, banking systems must be consistent, accurate, and able to handle large volumes of data. Currently, banks offer transaction support through various channels such as online banking, mobile banking, and in-person tellers. However, with the rise of digital technologies, the banking industry is increasingly turning to more and more sophisticated forms of transaction verification, while not compromising speed or authenticity. Whereas in decades past it may have taken days to complete a large banking transaction, in the modern age it can take mere seconds at most. With global and international requirements for these software systems being met, modern transaction software cannot rely on single servers, and must therefore rely on distributed systems not only because of speed and performance requirements but also due to reliability and accessibility issues; every few seconds a bank's trading server is down are potentially millions of dollars that go untraded.

1.2 Motivation & objective

The motivation behind our application is to implement a consistent, robust, authenticated, and reliable service that will allow banks to process transactions as efficiently as possible, for as many people as possible. Our objective is to develop a high-performance banking system, while also providing security measures to protect against fraud and unauthorized access. The system will be designed to handle large volumes of data. Overall, our objective is to provide banks with a reliable platform that will allow customers to process efficient transactions.

1.3 Overview of the application

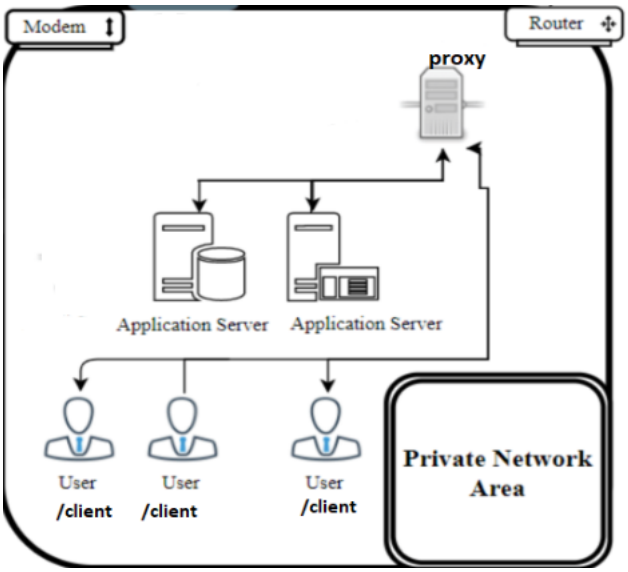
Our Software Development team designed and implemented a Distributed Banking System allowing for multiple servers servicing a bank to maintain autonomy and consistency of customer data in real time. This allows customers to perform transactions such as deposits, withdrawals, and money transfers between accounts, and view their account balance with as little latency as possible.



1.4 High-level architecture

Viewing the system at a high level is a simple client/server architecture.

The general overview of how the system would operate is as follows: the client would like to perform a transaction, so they must establish a connection to a banking server. The client proceeds to send a request to the "proxy" (or in our instance, an Nginx Reverse Proxy load-balancer which uses a round-robin algorithm), and they're forwarded to a server that is capable of doing the transaction. Once they've successfully connected to a server, they're able to start performing transactions. In our setup, there are three master servers, that will replicate all data actively. Only one master server is able to process write transactions (i.e. withdraw, deposit, transfer) on an account at any given point of time, by requesting locks for the account the client is attempting to modify. We will have one "read" server which will replicate data actively also. All the requests that do not require updating any accounts (i.e. view balance) will be redirected to our "read" server only, by our own "load-balancer" implemented in the Client process.



1.5 Implementation

A distributed system in a bank server system would likely implement replication to ensure that data is stored on multiple servers in order to provide redundancy and improve availability. Communication between the servers is achieved through WebSockets as they are an effective solution for communication in distributed systems due to their ability to facilitate bidirectional communication. Unlike traditional HTTP requests, which are unidirectional and require separate requests and responses, WebSockets allow for continuous interaction between the client and server. This real-time communication is particularly useful for updating and synchronizing data across different nodes in a distributed system

Synchronization is crucial for preserving the consistency and accuracy of data across all servers. It ensures that the data is current and valid on each server by implementing a locking mechanism, which we have achieved through the use of a static lock dictionary for accounts.

To maintain consistency, we ensure that all servers have an identical view of the data and that any updates made to it are promptly distributed to all servers. This is accomplished through the use of various consistency models, such as Sequential Consistency, which orders all write operations through our locking mechanism. Additionally, we have an algorithm in place to ensure that every account has the same balance. This algorithm checks the balance of each account on every server and instructs any server with a differing balance to update it to match the majority.

Fault tolerance is achieved through a combination of replication and redundancy, as well as the use of techniques such as failover, load balancing, and replication. This ensures that the system can continue to operate even in the event of a server failure or other issues. Our fault tolerance approach utilizes a "council of leaders" method, with three master servers coordinating with one another to ensure that the locking mechanism operates effectively and

eliminates single points of failure. In the event of a server not responding, the system employs a timeout mechanism to determine whether the server is dead or alive, and the decision to lock an account after a timeout is based on the local value of the lock on the remaining, active server.

2 Proof of Concept

Proof of Concept

Group #8

2.1 System Requirements

The client and server are being implemented in their own Java applications such that the client and server run their applications on their own end systems, and have the ability to communicate with each other via wireless/wired internet. The communication between the client and server applications will be facilitated via a persistent TCP socket connection. Additional per client and server end system requirements consist of:

- Client End System
  - Compatible with Java.
  - Send transaction requests to the server (Transfer, Deposit, Withdraw, View Balance etc.).
  - Receive responses from the server based on the success or failure of a transaction request.
  - May be intermittently connected.
  - May have a dynamic IP address.
  - Clients do not communicate directly with each other.
- Server End System
  - Compatible with Java.
  - Receive transaction requests from the client (Transfer, Deposit, Withdraw, View Balance etc.).
  - Send responses to the client based on the success or failure of a transaction request.
  - Precompiled SQLite Binary installed - database.
  - Always-on host.
  - Permanent IP address.
  - A single server should be able to handle multiple client requests as each client connection will be handled by a dedicated thread.
  - Multiple server end systems to simulate region-based servers.
  - Multiple servers to simulate a distributed system of servers where each server is responsible for servicing different geographical locations such that clients in different geographical locations can connect to the closest online server to.
  - Every server must ensure replication, data consistency/synchronization, and fault tolerance.

2.2 System Inputs

The users will enter their data/inputs in the Java console application, and the system will handle error/input checking based on the inputs provided by the users.

Initially, the users interact with the system by logging in or registering with the system. They enter username and password strings to log in to the system, which will either be accepted and log them into the system, or they will receive an error message if they entered an invalid username/password. If they do not have an account, users must register with a new username (that has not been taken) and password. After successfully logging in, the user will then be allowed to choose one of the following options:

- Withdraw money from an account
- Transfer money from one account to another
- Deposit money into an account
- Check the balance of an account

For withdrawing money, the user will enter the account number they want to withdraw from (has to be one of the user's accounts), and a decimal amount of money. If the account has more money than the amount requested, the money will be withdrawn from the account. Else an error message will tell the user they have insufficient balance to withdraw that amount from that account.

When a user wants to transfer money from one of their accounts to another, they must enter the account number which they want to transfer money from, and then enter the account number to transfer to, and lastly the amount they wish to transfer from account #1 to account #2. If account #1 has less balance than the amount requested to transfer is, the user will receive an insufficient balance error, or if one of the two accounts does not belong to the user, they will receive an invalid account error.

When a user wants to deposit money, they must enter the account number which they wish to deposit into and the amount they wish to deposit. If the account entered does not belong to the user or exists, they will receive an invalid account error.

To check the balance of one of their accounts, the user will only need to enter the account number of which they wish to see the balance. Once again, if the account number entered does not belong to the user or does not exist, they will receive an invalid account error.

2.3 Workflow of Data

2.3.1 Overview of Data Workflow

- One of the master servers receives the data request from the client (withdraw, deposit, transfer, view balance).
- The master server parses the account to be updated from the data request message that, and requests "lock" for the account (to prevent other servers from manipulating data related to that account, while it's processing the transaction for the client). This is done by broadcasting to all the other master servers to also lock the account that the broadcasting master server is wanting to lock.
- 2 cases: successfully lock the account, failed to lock the account.
  - Master server successfully locks the account:
    - Master server locks the account (meaning that all the other servers also successfully locked the account).
    - Processes transaction on the account.
    - Master server propagates the original data request message from the client to the other master servers, which still have the account locked.
    - The other master servers process the transaction on the account, then unlock the account locally and send a success message to the master server that received the master server that originally processed the client data request message.
    - The original master server that originally processed the client data request message unlocks the account.
  - Master server failed to receive the lock:
    - The master server will wait to acquire the lock for the account and continually attempt to do so until the lock is successfully obtained.

2.3.2 Explanation

The system receives the data from the user, which is initially processed by the master server, to verify that it is a valid operation in the required format. The data is stored in an SQLite database on all the servers.

The system retrieves data by querying the SQLite database on the master server, based on the type of operation being performed. For instance, during user login, the master server queries the database to verify the user's credentials. When a user carries out a transaction, the master server queries the database to obtain the account information and verify the account balances. Upon successful login, the master server communicates with the other master servers through web sockets, utilizing our locking mechanism, to determine if it is safe to proceed with the transaction.

When a user requests a transaction, the master server attempts to acquire a lock for the account by communicating with the other master servers. If it successfully obtains the lock, it will proceed to process the transaction(s). The master server then propagates the original data request message from the client to the other master servers that

still have the account locked. These servers will then process the transaction on the account, unlock it locally, and send a message indicating success to the master server that originally processed the client's data request. Finally, this server will verify that the balances are consistent across all servers and reply to the client with the updated balance for their account.

If the master server is unable to get the lock, the master server will store the record of the transaction(s) and keep on attempting and then apply them to the account once it receives the lock and updated account state.

In summary, the data is stored in an SQLite database within the master servers and is retrieved by querying the database. The local copy of the database is updated on each operation. The workflow of the data involves the master server attempting to get a lock for the account, processing the transaction(s), and sending the updated information to the other master servers once the lock is released.

2.4 Systems Guarantees

A distributed system is a system that is composed of multiple interconnected computers that work together to achieve a common goal. In our bank management distributed system, our severs would simulate servers that are spread across different locations and are connected by a network. The primary goal of our bank management distributed system is to provide seamless banking services to customers while ensuring that their data is secure and consistent across all nodes of the system and this system will also guarantee inter-country or inter-province manipulation operations on accounts (Deposit, Withdraw, Transfer).

2.4.1 Consistency of Updates and Replication

Consistency is one of the most important guarantees of our bank management distributed system. When a client performs a transaction, such as a deposit, transfer, or withdrawal, the system must ensure that the data is consistent across all nodes of the system. To achieve consistency, the system uses a master server mechanism, where the master server which connects to the client has to communicate with other master servers in order to obtain the lock associated with that account in order to process that transaction. When a master server finishes updating an account,it requests all of the other master servers to update the account in their local databases and then all other master servers send a release message to the initial server to which the connection was made by the client along with the updated balance of that account.

2.4.2 Characteristics for Consistency of Updates and Replication

- Atomicity: All updates to the data will be performed atomically, i.e., either all updates should be successful, or none of them should be applied.
- Isolation: Transactions will be isolated from each other which will be achieved using a locking mechanism as explained in the data workflow section. The changes made by one transaction will not affect the changes made by another transaction. This ensures that the data remains consistent and accurate which will be done using a locking mechanism.
- Consistency: The system will ensure that the data is consistent. This is achieved through replication, as described above.
- Durability: Once a transaction is committed, its changes should be durable and will not be lost, even in the event of a system failure. This is achieved through data persistence and backup mechanisms by the master server.
- Transparency: This system guarantees transparency as all the master server implementation is hidden and the client is not aware which server the request is reaching, This job is done by the load balancer.

2.4.3 Recovery In Case of Failure

Another important guarantee of our bank management distributed system is the ability to recover from system failures. In a distributed system, failure is not uncommon, and the system must be able to recover from failures quickly and efficiently.

To achieve this, the system should have the following characteristics:

- Fault-tolerance: The system is designed to handle failures and continue to operate even if one or more nodes of the system fail. If any of the master servers fail, the Nginx load balancer will redirect the transaction request to one of the other master servers. In the event that the read server fails, the load balancer will redirect the transaction request to one of the other master servers which are capable of processing read only requests (i.e. view balance).
- Redundancy: The system will have redundant copies of data and computation to ensure that the system can continue to operate even in the event of a failure.
- Recovery: The system will have mechanisms to recover from failures quickly and efficiently. This may include mechanisms to detect and diagnose failures, mechanisms to restore lost data, and mechanisms to restore the system to a consistent state. Most of these scenarios can be handled by our master server.

Each feature of a bank management distributed system, such as deposit, transfer, withdrawal, view balance, etc., should have the same consistency and recovery guarantees as described above. Additionally, each feature may have specific requirements, such as ensuring that transactions are performed in a timely manner or that data is protected from unauthorized access.

For example, the transfer feature should ensure that funds are transferred from one account to another atomically, i.e., either the entire transfer is successful, or none of it is. The view balance feature should ensure that customers can view their account balances accurately and securely, without the risk of unauthorized access.

2.5 Evaluation

On an application level, our implementation of the banking system should be capable of:

- Manipulating account balance (Deposit, Withdraw, Transfer).
- Account creation and verifying account login information.
- Maintaining an accurate, stable, and reliable record of everyone's banking information.
- Fast communication between servers.

On a system level, our design specifications will be sufficient for our implementation of these features.

2.5.1 Manipulating account balance (Deposit, Withdraw, Transfer)

Manipulating the account balance is a bit complicated, when it comes down to the master server that received the original data request message convincing the other master server(s) that the operation on the account in question did in fact occur. We achieved this by implementing an account-locking mechanism. In this approach, once a client requests to do a transaction, the master server that received the original data request message will check its local static dictionary to see if the account it is trying to process a transaction on is locked or not. If the account is not yet locked, this master server will reach out to the other master servers to receive the lock for the account. Once it receives the lock, the master server who connected with the client can process the transaction. Finally, when the transaction has been completed, the server releases the lock (by informing the other master servers) and sends every server the client data request message so that every server can also apply the update to their local database instances while the account is still locked. Once the transaction is applied on all servers, all servers release the lock. In the event that the account is locked, the master server will wait to acquire the lock for the account and continually attempt to do so until the lock is successfully obtained. This way, we ensure that all transactions occur in the same order on all servers, since our locking mechanism guarantees the order of WRITE operations, giving our system sequential consistency.

2.5.2 Viewing an account balance

Our distributed system has a separation of concerns such that all read operations (i.e. view balance) are processed by a dedicated read server, and all write operations (i.e. withdraw, transfer, deposit) are processed by one of the three master servers. Viewing an account balance does not require any manipulation of a users account information, thus, all read requests are redirected through a load balancer that we have implemented in our client process to the dedicated read server. The read server receives the clients data request message, parses and verifies the account in the message, and then uses that account to retrieve the account balance from the database and send it back to the client for viewing.

2.5.3 Account creation and verifying account login information

Creating an account and verifying login information is distinct from manipulating account balances, as no locks are needed as we do not write into balance of any existing account. We simply validate user credentials against the database, allowing the client to proceed with their desired transactions. Moreover, when a user signs up as a registered user, we only need to create a new user entry in the database, assign them an account, and synchronize the registered user information across all other databases with their respective master servers. These processes can be carried out without requiring any locks, ensuring smooth and seamless execution.

2.5.4 Maintaining an accurate, stable, and reliable record of everyone's banking information

To argue that this is possible with our current system design we will argue that our system is capable of upholding accuracy (low error rate), stability (unchanging data), and reliability (high accessibility/uptime percentage). This can be argued by demonstrating that both the transmission and storage of data uphold these parameters. Reliability is probably the easiest to justify since we intend to have a high number of redundant servers which will themselves have a high uptime percentage, and our data storage is local to those "reliable" servers, and thus our system as a whole will achieve "reliability". For accuracy and stability of the data storage, this comes naturally with our storage choice of being local to the master servers; the servers themselves will not read/write account data or move data around unless explicitly asked to by a client, thus maintaining accuracy and stability. For upholding accuracy and stability over transmission, we rely again on our system design. Firstly, the system's design involves ensuring that the master server(s) always have up-to-date information regarding all the accounts. This ensures that any server processing transactions is working with accurate and up-to-date data, reducing the risk of errors or inconsistencies. Secondly, the system's design involves a locking mechanism to prevent multiple servers from modifying the same account simultaneously. This prevents conflicts and ensures that transactions are processed in a consistent and orderly manner, reducing the risk of errors or inconsistencies. When a server is finished processing a transaction, meaning all of the other servers are also finished processing a transaction, all servers release the lock. This ensures that the master servers always have the most up-to-date account balances, ensuring accuracy and stability, and ultimately sequential consistency.

3 Replication

Replication

Group #8

3.1 General

The general flow for this milestone (replication) will be implemented using communication between master servers, which will implement a locking mechanism to prevent race conditions. The client will send a request to the server, which then the server will communicate with other master servers to obtain the lock for the account so it may proceed with the request. Once the lock is obtained, the server will execute the request, and then communicate with other master servers with the user request to allow them to update their databases to ensure proper replication and synchronization. We have implemented three master servers for this milestone.

After all servers have completed their updates and sent back their final values, the requesting master server will verify that all the final values match. In the event that there are mismatches, the master server will use an algorithm to determine the majority value and then communicate with the servers that provided mismatched values to update their account values accordingly.

This verification step is crucial to ensure that all servers have properly replicated and synchronized their data. By verifying that all servers have the same final values, we can be confident that our replication system is working as intended.

Overall, this communication and locking mechanism, along with the final value verification step, will allow us to ensure that all data is properly replicated and synchronized across all master servers. This will help to ensure the consistency and reliability of our system.

3.2 Lock

When a user request arrives at a master server, we've identified three cases in which to categorize and process the request. These cases are generalized to any number of master servers and can easily be applied to three master servers, as our current implementation requires:

3.2.1 Case #1: No master server has a lock for the account

In this scenario, the master servers communicate with each other and determine that no other master server currently has a lock for the account. Since there is no lock, the master server creates a lock for the account and communicates with the other master servers that this account has a lock. The transaction can then be processed by the server. After the transaction has been fully processed, the lock is removed from the account, all master servers are informed of the transaction and update their databases, and other transactions can occur on the account now.

3.2.2 Case #2: Exactly one master server has a lock for the account

In this scenario, the master servers communicate with each other and determine that exactly one other master server currently has a lock for the account. In this case, if another transaction occurs for the specific account, the server will communicate and request a lock from other master servers, and see there is already a lock for this account. The server will then wait until the lock is released in a while loop, and once the server obtains the lock for the account, it will process the user transaction. Once processed, then all master servers remove the lock from the account, and all master servers are informed of the transaction and update their databases. Other transactions can occur on the account now.

3.3 Data

The data (ie withdraw, transfer, deposit) sent by the client is sent to the server, where this master server will communicate with other master servers to ensure there is no lock for the account and the transaction can proceed. Once the transaction has been fully processed, the lock is released and all master servers will process the same transaction to ensure replication and synchronization on each of their databases.

4 Fault Tolerance

Fault Tolerance

Group #8

4.1 Code Summary

The fault tolerance mechanism in our system is based on the "council of leaders" approach, where in our case we have three master servers. When a client connects to one of the master servers (let's call it Server A), server A initiates separate connections with the other master servers (call them Servers B and C) and opens a new socket for communication.

In the event of a failure of the "read" server, the load balancer will redirect any requests to "view the balance" to one of the available master servers.

To ensure that a lock for an account is only given away when all three master servers agree, server A waits for responses from all other servers, in this case server B and server C before giving away the lock. If server A doesn't receive a response from a particular server within say, 5 seconds, it assumes that either server B or server C is down and catches this error in a *SocketTimeoutException*. In this case, server A sets the response to true (for that particular server that is considered down), allowing it to give away the lock without waiting for an actual reply from the other servers. Generally, we have indicated two high-level cases of server response:

4.1.1 Case #1: Both servers respond.

In this case, server A receives the status of the lock from both servers and can make a decision on whether to issue the lock or not.

4.1.2 Case #2: At least one server does not respond.

This case is treated symmetrically across all non-responding servers; if server A does not receive a response from a server say, server B, we'll call it a non-responding server, and server A assumes that server B is down and sets the response to "true" if the local value of the account is still "false" meaning no other server asked for the lock of the same account and then assume the response to be "true". It then gives away the lock, assuming that all non-responding servers have not already given away the lock for that account.

If server A goes down while a client is connected to it, the request is redirected to another server, in our case either server B or server C, with the help of the load balancer. However, there are three possible scenarios for this:

4.1.2.1 Case A: Server A failed before initiating the locking process.

In the event of a server failure prior to initiating the lock process, the load balancer can effortlessly redirect the traffic to one of the functioning servers, either server B or server C, through its load-balancing capabilities.

4.1.2.2 Case B: Server A fails after initiating the locking process.

In the case of a server failure after initiating the locking process, it is necessary to forcibly unlock the account on either server B or server C. This is achieved through a timeout mechanism, where if the server does not receive an unlock message within 5 seconds of receiving the lock message, it will automatically unlock the account without conducting the transaction.

4.1.2.3 Case C: Server A fails after unlocking the account and completing the transaction.

If server A experiences a failure after successfully unlocking the account and completing the transaction, the load balancer can easily redirect the connection to either server B or server C. These servers only need to return the updated balance information, as the transaction has already been executed.

4.2 Database Recovery

Fault tolerance is a critical aspect of system design that ensures that a system can continue to operate in the event of hardware or software failures. In a server environment, database management is a key component of fault tolerance as it provides a central repository for storing and retrieving data. Our team had originally planned to implement a fault tolerance strategy for database recovery where the working server would send transaction history to the down server upon recovery. However, we have since realized that a better approach would be to implement database synchronization by having the server that is down, request an updated database upon reboot. This approach is simpler, faster, and more scalable, and it will allow us to efficiently manage our databases and reduce downtime. We believe that this approach to fault tolerance and database synchronization will lead to a more resilient system and ultimately benefit our operations. This hasn't been implemented within this iteration, as we aim to implement it for the synchronization demo.

5 Synchronization & Consistency

Synchronization & Consistency

Group #8

5.1 General Synchronization & Consistency

To achieve synchronization, the following mechanism is implemented. When a client initiates a transaction with a server, call it server A, the server will perform the transaction and inform all other servers, call them servers B and C, of the transaction. Servers B and C will then process the transaction and report to all other servers the new balance for the account in question. This process ensures that all servers have processed the transaction and have the same data for the account. While the example given involves three servers, this mechanism can be generalized to any number of servers with the following case structure:

5.1.1 Case #1: All servers agree on the new balance.

This is the ideal scenario where all servers have processed the transaction and arrived at the same new balance for the account. In this case, the transaction is considered complete, and nothing further needs to happen. The client can be notified of the transaction's success, and the system can move on to the next transaction.

5.1.2 Case #2: Majority of servers, but not all, agree on the new balance.

This is somewhat of an extension to Case #1. In this case, the majority opinion is accepted, and the servers that disagree will update their value to match the other two. The reason for this is to ensure that all servers have the same data and are in sync with each other. The server that disagreed may have experienced a network issue or a hardware failure that caused it to process the transaction incorrectly, and by updating its value to match the other two the system can maintain consistency across all servers. Once the update is complete, the system can proceed with the next transaction.

5.1.3 Case #3: Majority of servers disagree on the new balance.

This scenario should never happen, and it's an indication that something has gone wrong with the synchronization mechanism. It could be a sign of a bug in the system, or it could be due to a hardware failure or a network issue that caused the servers to process the transaction differently. To resolve this issue, the system would need to investigate the cause of the disagreement and take corrective action to ensure that all servers have the same data and are in sync with each other. Until the issue is resolved, either manually or automatically, the system will not proceed any further with any transactions.

5.2 Database Recovery

In the event that a server goes down, it is crucial to ensure that the data on that server is not lost and that the server can recover its data once it comes back online. To achieve this, the system is designed to store a copy of the database on each server.

When a server goes down and comes back online, it will request an "up-to-date" copy of the database from another server. This ensures that all servers have the same data and are in sync with each other. Once the server has received the database, it can then proceed with any transactions it needs to process.

To add to the previous paragraph, it's important to note that the process of requesting an "up-to-date" copy of the database from another server and then applying it assumes that no other transactions occur during this time. In reality, this is not always the case. If a transaction occurs on the database while the updated copy is being sent to the server, the server that requested the updated database may apply the transaction before applying the "up-to-date" database to its own database. This can result in inconsistent data between servers and can cause issues with data integrity in the real world. However, for this project, our approach seems to be sufficient as we have a majority consensus occurring (as discussed above) that should prevent issues from this implementation.

5.3 Summary

In summary, the synchronization mechanism in a distributed system such as the one described is crucial to maintaining consistency across all servers. The system is designed to handle different scenarios that could arise during

synchronization, such as when two out of the three servers agree on the new balance. Additionally, the system is resilient to server failures and can recover data from other servers, ensuring that all servers have the same data and are in sync with each other.